Data Sci. Eng. (2017) 2:169–177
DOI 10.1007/s41019-017-0039-z

Low-Overhead Paxos Replication

Jinwei Guo[1] · Jiajia Chu[1] · Peng Cai[1] · Minqi Zhou[1] · Aoying Zhou[1]

Received: 30 November 2016 / Revised: 10 March 2017 / Accepted: 13 March 2017 / Published online: 22 March 2017

Abstract Log replication is a key component in highly available database systems. In order to guarantee data consistency and reliability, it is common for modern database systems to utilize Paxos protocol, which is responsible for replicating transactional logs from one primary node to multiple backups. However, the Paxos replication needs to store and synchronize some additional metadata, such as committed log sequence number (commit point), to guarantee the consistency of the database. This increases the overhead of storage and network, which would have a negative impact on the throughput in the update intensive work load. In this paper, we present an implementation of log replication and database recovery methods, which adopts the idea of piggybacking, i.e., commit point can be embedded in the commit logs. This practice not only retains virtues of Paxos replication, but also reduces disk and network IO effectively. We implemented and evaluated our approach in a main memory database system. Our experiments show that the piggybacking method can offer 1.3× higher throughput than typical log replication with synchronization mechanism.

Keywords Log replication · Database recovery · Paxos · OceanBase

✉ Peng Cai
pcai@sei.ecnu.edu.cn

[1] East China Normal University, 3663 N. Zhongshan Rd., Shanghai 200062, China

1 Introduction

Through the smart phone, we can submit transaction processing requests to the databases at any time. And in the scenario of Internet application, highly concurrent requests have overwhelmed the traditional database systems. For example, in Chinese "Single Day" (i.e., Double 11 shopping carnival), the total transactions may hit the level of hundreds of millions in the first minute. To resolve this challenge, many NoSQL and NewSQL systems were designed and implemented [7]. NoSQL refers to the data storage systems which are non-relational, distributed and not guaranteed to follow the ACID properties. Compared to the relational DBMS's, NoSQL systems have some excellent characteristics, such as without needing to predefine the data schema, high scalability, share-nothing architecture and asynchronous replication. These features provide strong support for the Internet applications in the Web 2.0. On the other hand, both the industrial and academic communities hope to use the unique features of NoSQL to solve the massive data processing problems. NoSQL systems have got extensive attentions, and main industry players including Google, Amazon and Facebook have developed their NoSQL database products which have played a key role in their services.

NoSQL systems have some limitations when they are used in the mission critical applications which require strong data consistency. For example, asynchronous replication and the eventual consistency mechanism provided by NoSQL are not applicable for the bank systems. If the delay of inconsistency window is too long and the primary crashes in this delay period, then the last update information may be lost because the committed update transactions have not been synchronized to the backups. In this procedure, it is possible that a customer performs a withdraw

operation, but the final balance of the account is not reduced accordingly.

Log replication based on Paxos [15] can achieve the strong data consistency. The Paxos algorithm is proposed by Leslie Lamport in 1990 which is a consistency algorithm based on the message passing model. The algorithm solves the problem of reaching agreement among multiple processes or threads under the distributed environment. Recently, there exist many systems adopting Paxos algorithm to the log replication [1, 10, 24, 25]. As long as the log records have been replicated in the majority of servers, the primary node can commit the transaction. This method can guarantee the strong consistency between primary and secondary nodes. When the primary node failed, the majority of the system nodes can select at least one and only one new primary to achieve a seamless takeover of the predecessor.

Unfortunately, a typical implementation of Paxos replication—which adopts two-phase commit protocol (2PC) using some metadata such as commit point to guarantee the consistency of the database—can increase the overhead of disk and network. In other words, the synchronization of metadata for consistency can put an excessive burden on the disk and network, which causes negative impacts on the throughput of the transactional database. Therefore, this paper presents a low-overhead log replication, which adopts the thought of piggybacking. More precisely, the commit point is embedded in the transactional log and then is synchronized from the primary node to backups along with the log records. We implemented this mechanism on an open source database system OceanBase [19] and showed that this method can provide good performance in terms of throughput of transaction processing since the overhead of disk and network was reduced.

The remainder of the paper is organized as follows: Preliminary works, which include the OceanBase architecture and Paxos replication model, are presented in Sect. 2. We introduce related work of Paxos replication in Sect. 3. Sections 4 and 5 introduce the mechanism of log replication and recovery for OceanBase, which aims to reduce the overhead of disk and network. Section 6 presents experimental results. We conclude the paper in Sect. 7.

2 Preliminary

In this section, we will introduce the database system OceanBase, where our low-overhead method is implemented. And then we will describe and analyze the mechanism of typical Paxos replication.

2.1 OceanBase

OceanBase is a scalable relational database management system developed by Alibaba. It supports cross-table and cross-row transactions over billions of records with hundreds of terabyte data.

OceanBase can be divided into four modules: the master server (RootServer), update server (UpdateServer), baseline data server (ChunkServer) and data merge server (MergeServer).

- *RootServer* It manages all servers metainformation in an OceanBase cluster, as well as data storage location.
- *UpdateServer* It stores updated data in OceanBase. UpdateServer is the only node responsible for executing any update requests such as `DELETE` or `UPDATE` SQL statements. Thus, there is no distributed transaction in OceanBase because any update operations are processed in a single node.
- *ChunkServer* It stores OceanBase baseline data, which is also called static data.
- *MergeServer* It receives and parses SQL requests, and forwards them to the corresponding ChunkServers or UpdateServer after lexical analysis, syntax analysis, query optimization and a series of operations.

UpdateServer is a key component in OceanBase, and it has some characteristics, which we utilize to implement our Paxos replication, as follows:

- UpdateServer can be seen as a main memory database, which stores updated data in memtable.
- One transaction only corresponds to one commit log, which is generated when the transaction is finishing.
- Log records are stored on disk continuously. Therefore, there are no holes in log files.

OceanBase can be configured with multiple clusters, e.g., one master cluster and one slave cluster. Only the master can receive write requests and process these transactions. When master cluster breaks down, the whole system is not available for clients. For this reason, we have implemented Paxos replication, whose model will be introduced in the next subsection.

2.2 Paxos Replication Model

Using Paxos to replicate log records is a popular choice to build a scalable, consistent and highly available database. Traditionally, systems adopting Paxos replication have two main phases: Leader election and log replication. The servers participating in these phases are called Paxos members, which make up a Paxos group. For ease of description, we can use *member* to refer to the member of Paxos group.

During the Leader election phase, there may be no Leader in the system. Assume that each member in the Paxos group should take part in the Leader election. Therefore, they report the local last log sequence number (LSN) to the election service. Note that the local last LSN may be comprised of log id, generated log timestamp or epoch number, which can be used to order the state of each member. The election service elects a Leader from the reporting members in consideration of the LSN's, i.e., the new Leader's LSN must not be less than a majority of members'. When a majority of nodes acquire the election result and succeed in registering to the new Leader, the election phase is finished successfully. It is noteworthy that the Leader can maintain its authority by renewing its election lease periodically. If the majority of members note that the Leader's lease is expired, the system will enter into the Leader election phase again.

During the log replication phase, each member has one of the two replica status: Leader and Follower, which own primary and backup replica, respectively. As is shown in Fig. 1, the Leader receives a write request from a client, generates a commit log, and replicates the log record to all Followers. When a Follower receives the log message from the Leader, it can do different actions according to the strategies of reliability:

- *Durability* The Follower cannot response the Leader until they ensure that the log record is persistent in local disk.
- *Non-durability* The Follower responses the Leader immediately when it receives the log message and buffers the log record in memory.

The execution flow of point 1 in Fig. 1 shows the situation of non-durability. We note that the delay of write request is smaller than the flow of point 2, which means the situation of durability.

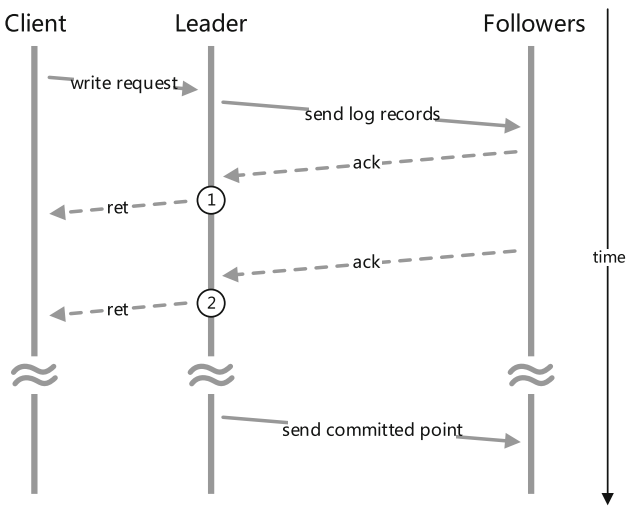

Fig. 1 Log replication model based on Paxos

When the Leader gets a majority of acknowledge responses, it can update local committed LSN, which can be called *commit point*. Then the Leader flushes the point to disk and sends it to each Follower. Note that the commit point is an important metadata in Paxos replication, which is used to guarantee data consistency for read operation and recovery. In other words, this metadata enables the Follower to provide clients with timeline consistency services, and simplifies recovery processes which guarantee the data consistency when the database recovers from a failure. However, the traditional synchronization mechanism of commit point increases the overhead of storage and network. Therefore, we design and implement the low-overhead log replication adopting durability strategy for OceanBase.

3 Related Work

ARIES [18] has been the actual criteria for transactional logging in traditional database systems. It gives a reference for log model and recovery mechanism. Replication, such as eager or lazy mechanism [13], is an effective means to provide horizontal scalability, high availability and fault tolerance in distributed systems. Therefore, it is common for distributed database systems to replicate transactional log from a primary node to one or multiple backup replicas.

As the Internet applications are developed rapidly, an increasing large number of databases leverage the NoSQL techniques, which provides us with scalability and high availability through the use of replication. Dynamo [11], which is developed by Amazon, is a highly available key-value datastore system. Its replication resorts to NWR strategy, which permits clients to decide to balance availability against consistency. Facebook's Cassandra [6, 14] adopts the idea of Dynamo and makes use of the optimized mechanisms, such as load balance. At present, it has became an open source distributed database management system in Apache. Yahoo's PNUTS [8] is a scalable datastore, which is focused on cross-datacenter replication. Although these systems can offer good performance and high availability, the eventual consistency can be only provided.

Paxos is a consensus protocol for solving consistency problem in distributed systems. It is described basically by Lamport in [15]. Multi-Paxos introduced in [16] is an important protocol for Paxos replication. And more variants are introduced by him in [17]. Using Paxos for replication is a common choice for implementing scalable, consistent, and highly available datastore. Chubby [5] is Google's service aiming at providing a coarse grained lock

service for loosely coupled distributed systems. ZooKeeper [26] is its open source implementation. Google has developed MegaStore [1], Spanner [10] and F1 [24], and these database systems have used Paxos for log replication. Megastore is a storage system providing strong consistent. Spanner is a scalable, multi-version, global distributed, synchronous replication database. F1 provides the functionality of the SQL database. Raft [20] is a consensus algorithm for RAMCloud [21]. It is designed to be easy to understand and equivalent to Paxos. Rao et al. [23] introduce a relatively complete technology solution to build a datastore using Paxos. Patterson et al. [22] analyze and discuss the replication based on Paxos.

In recent years, with the rapid development of new hardware, e.g., SSD (solid-state drive), NVM (non-volatile memory) and RDMA (remote direct memory access), new log replication and recovery mechanisms emerge. Dragojević et al. [12] leverage NVM and RDMA to build a highly scalable and available computing platform without sacrificing the strong consistency. Tango [2, 3] and Hyder [4] use log-sharing architecture—which is based on SSD and high-speed network—to ensure the reliability and availability.

4 Low-Overhead Replication Protocol

This section will describe the log replication protocol based on piggybacking method, which will reduce disk and network IO. To simplify the discussion, we adopt three-way replication. More precisely, the Leader—which is the primary node—replicates log records to two Followers owning the backup replicas.

Recall that the architecture of OceanBase is described in Sect. 2.1. In order to give a simple explanation, we treat MergeServer's and ChunkServer's as the clients which forward write requests, UpdateServer's and RootServer's as the Paxos members in log replication model mentioned above, which are responsible for log replication and Leader election, respectively.

4.1 Commit Log Entry

In the Leader's term, the primary replica is the only one which can receive and process the write operations. When a client issues a write request, it first acquires the information about which UpdateServer is the Leader, and then sends the write to the Leader node.

When the Leader receives a write request, it gets the corresponding transaction, processes the operation and pre-applies the results to the local memory table. Once the transaction enters the commit phase, a log entry—which contains a unique LSN, the transaction ID and the log data (the final results produced by the transaction)—will be generated.

Recall from Sect. 2.2 that the commit points need to be replicated and persisted in Paxos members, which can be achieved by the synchronization of log records. Therefore, in order to reduce the impact of handling commit points, the committed LSN can be embedded into each commit log record. The format of the log entries carrying the committed LSN is shown in Fig. 2. We note that the commit log entry with LSN of 101 holds a committed LSN of 99, which indicates that the logs whose LSN is not greater than 99 can be applied to the memtable.

It is clear that embedding the commit point into the log increases the size of log. Since the size of one LSN is only a few bytes, the extra information will not take up too much space. The group commit mechanism which will be described below can further reduce the impact of embedding the metadata.

4.2 Log Replication

After generating the commit log, the Leader needs to send the log record to all Followers by an asynchronous network function, which does not block the single commit thread held responsible for synchronization and persistence of log records. In other words, the Leader is able to flush the commit log to local disk without waiting for the responses from the Followers.

When a Follower receives a log message from the Leader, it would extract the committed LSN from the received log record and compare it with the local cached committed LSN. If the local value is less than the new committed LSN, it should be updated with the new one; otherwise, the value is not refreshed. Then the Follower appends the commit log to the end of the log file in local disk. Once the appending operation has finished or overrun a certain period of time, the Follower would get the maximum flushed LSN, which represents the latest state of commit log in local disk. Then it sends the response

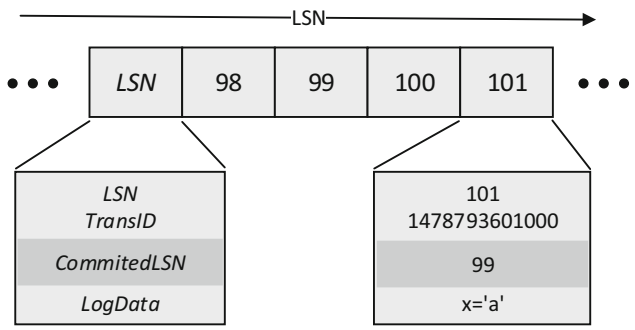

Fig. 2 An example of commit log entries with committed LSN

message containing the maximum flushed LSN to the Leader.

The value of committed LSN is checked periodically by the log replay threads in the Follower node. When the value is changed, these threads will get and replay the persistent log which has not been replayed in local. More precisely, the log entries whose LSN's are not greater than the committed LSN need to be applied to the local memory table in order of LSN. If the Follower cannot find the corresponding log records in the log file, it will request the missing ones to the Leader by themselves.

In order to compute the commit point, the Leader has to store the flushed LSN's of all the Followers. When receiving the response message from a Follower, the Leader would extract the flushed LSN from the message and compare it with the local cached value of this Follower. If the new flushed LSN is greater than the local one, the Leader will replace the local value; otherwise, the value is not changed. Based on all the Followers' flushed LSN's cached in local and the majority strategy, the Leader calculates a new committed LSN, which indicates a certain state of the database. If the new value is greater than the previous one cached in local memory, the local committed LSN will be updated with the new one and embedded in the next commit log entry. Finally, the Leader commits the corresponding transactions in accordance with the local cached commit point and returns the results to the clients.

Figure 3 shows an example of Paxos members' information stored in Leader, which is used to compute the committed LSN. We find that the log records whose LSN is not greater than 120 are durable in majority of members. Therefore, the Leader can update the committed LSN to 120, commit the transactions whose LSN's are not greater than 120, and embed the value 120 into the next commit log entry. If a Follower receives the new commit point, it will apply the logs to local memtable in order of LSN until 120.

Note that the members' information is only stored in Leader's memory. Therefore, a new Leader needs to request the flushed LSN to the Followers when it starts to take over the replication. And if a Follower fails, the primary node needs to clear out the failure's information from the table.

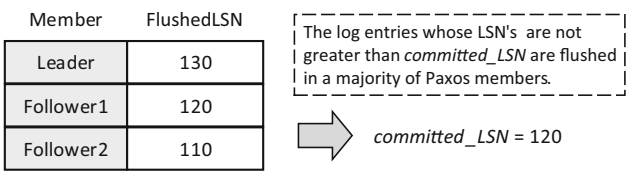

Fig. 3 An example of Paxos members' information used to generate committed LSN

4.3 Further Discussion

The main procedure of the log replication protocol has been introduced above. However, we note that the synchronization triggered by each commit log entry can produce massive disk and network IO operations. Therefore, in order to further reduce the overhead of log replication, it is common for many databases to adopt group commit mechanism, which treats a group of log records as one commit unit.

When the Leader generates a log record for a transaction, it caches the log record in the local buffer. Once the size of log in the buffer reaches the maximum capacity or the time interval from the last successive commit is longer than a configured value *commit_interval*, the Leader packages the buffered log entries and sends the package to each Follower by an asynchronous method, and then appends all of them to the log file in disk. Therefore, we can generate a special commit log entry containing committed LSN at the end of group, which can reduce the space overhead in the log.

However, it is difficult to configure *commit_interval* since the hardware may be not same in different production environments and the processing time of log replication in Followers should be considered. Therefore, we can design an adaptive group commit mechanism, which takes into account the time of log persistence in each replica node. Let $persistence\_time(i)$ denote the latest time of flushing a group of logs in node i. When the Leader receives the $persistence\_time(i)$, it needs to recalculate the *commit_interval* as follows:

$$commit\_interval = (commit\_interval + persistence\_time(i))/2 \quad (1)$$

The *commit_interval* is initially set to a value configured by the administrator, and it is automatically changed to a relatively stable value—which is suitable for the platform of database—through the simple adaptive method. If a replica node i encounters something abnormal, the Leader will catch the exception and not consider the value $persistence\_time(i)$.

5 Recovery

In this section, we describe how a Paxos member recovers from a failure. It is clear that the failure is a common phenomenon in distributed systems, e.g., power failure, administrator mistakes, software or hardware errors and so on. Therefore, we need to adopt recovery mechanism to ensure the correctness and consistency of the database.

Recall from Sect. 2.2 that the system adopting Paxos replication has two phases. Therefore, each Paxos member needs to periodically check the system status, which can be presented by a local variable. There are two kinds of states of the status, i.e., DURING_ELECTION and AFTER_ELECTION, which indicate whether there exists a Leader in the system. If the status is AFTER_ELECTION, it shows that the system is in the log replication phase. When a member is restarting, its election role is definitely determined. Therefore, it can take predetermined actions in accordance with the role. If the system is in DURING_ELECTION state, it means that the Paxos group is in the Leader election phase. The recovering member needs to take part in the election. It is only when new Leader is elected that the restarting member can continue to recover.

5.1 Leader Recovery

When a recovering member finds the status of system is AFTER_ELECTION and its election role is Leader, it has to take some steps to ensure the consistency of the database. In other words, it is not until the new Leader guarantees that its local log records are persisted in a majority of the members that it can service requests from the clients. Firstly, it scans the log file in disk, gets local last LSN and max committed LSN from the file, and caches them in local variables. Then the Leader starts up threads to replay whole local logs from checkpoint. By this time, the Leader cannot service any requests from clients. Next, it appends a special commit log entry which only contains max committed LSN, and replicates the record to other members. Finally, the master receives the responses of Followers and updates corresponding information of the servers. When the primary replica detects that the committed LSN is not less than the previous cached local last LSN, it can provide services for clients.

Procedure 1 Leader Recovery

```
1: local_last_LSN = the last LSN from local log files;
2: committed_LSN = the max committed LSN from local log files;
3: start up threads to replay local log;
4: push a NOP task to commit queue;
5: while local_last_LSN > committed_LSN do
6:     temp_committed_LSN = get committed LSN from Follower responses;
7:     if committed_LSN ≤ temp_committed_LSN then
8:         committed_LSN = temp_committed_LSN;
9:     end if
10:    sleep for a while;
11: end while
12: state = ACTIVE; // can provide service
```

The Leader adopting the above recovery procedure can guarantee the consistency of the database. Its main steps are summarized in Procedure 1. If the Leader has taken over the log replication completely, a client getting the data from the primary replica can be provided with strong consistency. In some cases, we would like high availability rather than consistency. Therefore, when the Leader starts the replaying task, it can service read requests from clients as long as replaying local log to the committed LSN. The remainder of the local log is applied along with the new committed LSN.

5.2 Follower Recovery

When a restarting member finds the status of system is AFTER_ELECTION and its election role is Follower, it has to ensure that the state of local data is consistent with the Leader. Since the Follower cannot judge whether the log records whose LSN is greater than the committed LSN should be applied to the local memory table, it must get necessary information from the Leader. In order to reduce the network overhead, we implement a recovery mechanism as below. To begin with, the Follower scans log file in disk to update local variables, e.g., local last LSN, committed LSN. As described above, the committed LSN is the max committed LSN stored in the log file. Then, it starts to replay local log records whose LSN is not greater than the committed LSN, and it discards the remaining log records. At the same time, the Follower reports its committed LSN to the Leader. When the Leader receives this message, it sends the corresponding log records after that LSN to the Follower. Finally, the Follower can receive new log records and refresh the committed LSN, which triggers itself to replay the log continuously.

Note that if the role of Leader is frequently switched in different members, the log records of committed transactions will be lost. To prevent this, it is not until the backup node ensures that the received log records which are integrated from the committed LSN and whose LSN's are greater than the local last LSN that Follower can discard log records after the committed LSN. In other words, the Follower buffers the new log entries until these data cover the LSN range $(local\_committed\_LSN, local\_last\_LSN]$, and then replaces the corresponding log entries in disk atomically. The main steps are illustrated in Procedure 2.

Procedure 2 Follower Recovery

```
1: local_last_LSN = the last LSN from local log files;
2: committed_LSN = the max committed LSN from local log files;
3: local_committed_LSN = committed_LSN;
4: start up threads to replay local logs to committed_LSN;
5: register and report local_committed_LSN to the Leader;
6: while local_last_LSN > master_LSN do
7:     master_LSN = the max LSN of continuous logs from the Leader;
8:     sleep for a while;
9: end while
10: discard log records after local_committed_LSN;
11: append new logs to log file;
12: state = ACTIVE; // can response to Leader
```

After the Follower applies the commit log entries—whose LSN's are not greater than the

local_committed_LSN—to the memory table, it can provide clients with weakly consistent services, such as timeline or snapshot consistency. Therefore, a client forwards a read request to a Paxos member in accordance with the requirement of consistency.

6 Experiments

This section evaluates the performance of several different implementations of the synchronization of the commit point, i.e., piggybacking method, synchronization method and asynchronous method, which are implemented in the open source database system OceanBase 0.4.2:

- *Piggybacking method (PIGGY)* This method is our implementation in this work described above. In order to reduce the disk and network overhead, we append a special commit log entry containing the committed LSN to the end of the log group.
- *Synchronization method (SYNC)* This method is different from PIGGY. When the Leader detects that the committed LSN has been changed, it would call Linux interface `fsync()` to flush the committed LSN to the disk and call a method which sends the commit point to the Followers asynchronously.
- *Asynchronization method (ASYNC)* This method is different from the above two methods. The Leader starts a background thread, which is responsible for flushing and sending the committed LSN periodically.

All the methods adopts the group commit technique introduced specifically in Sect. 4.3.

6.1 Experimental Setup

This subsection describes the platform of the cluster and deployment of the database, and gives a brief overview of the benchmark.

Cluster Platform We ran the experiments on a cluster of 18 machines. The software and hardware setup of each server is shown in Table 1. Note that the write latency of the SSD is about hundreds of microseconds.

Table 1 Experimental setup

Software and hardware setup	
CPU	E5606@2.13 G * 2
CPU cores	8 (Hyper-threading disabled)
Memory	16GB PC3L-12800R * 6
Disk	100 GB SSD * 1
Network	Gigabit Ethernet
Operating system	CentOS 6.5

Database Deployment The database system is configured with three-way replication. More precisely, we start three OceanBase clusters, and each instance—which contains a Paxos member (RootServer and UpdateServer) running in one node and 5 clients (MergeServer and ChunkServer) in others—is deployed on 6 servers.

Benchmarks We adopted YCSB [9]—a popular key-value benchmark from Yahoo—to evaluate the log replication performance of the three methods, we used workloads containing heavy replace operations with read/write ratio of 0/100. Since the MergeServer is the external interface of the database system, the application of YCSB should connect to MergeServer firstly, and then executes *replace* auto-commit transaction repeatedly. We observe the results of YCSB after the execution and the statistics of system during the execution. The database is preloaded with 10 million records, and the size of each record is about 100 bytes.

6.2 Log Replication Performance

We ran experiments to benchmark PIGGY against the other methods by using the YCSB. The performance in the terms of *TPS, IOPS, write throughput* and *Follower receiving* was focused on. The *TPS* refers to the number of transactions performed by the system per second. The *IOPS* is used to denote the write requests issued to disk per second. Let *write throughput* and *Follower receiving* represent the volume of data flushed to disk in the Leader and the number of messages received in Followers over a period of time. The experimental results are illustrated from Figs. 4, 5, 6 and 7.

We first compared PIGGY with other methods for the *TPS* case, which reflected the performance of transaction processing while executing various workloads. Figure 4

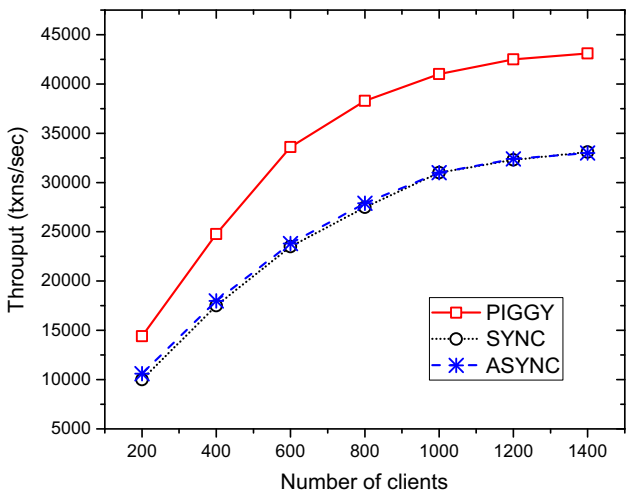

Fig. 4 Throughput of transactions

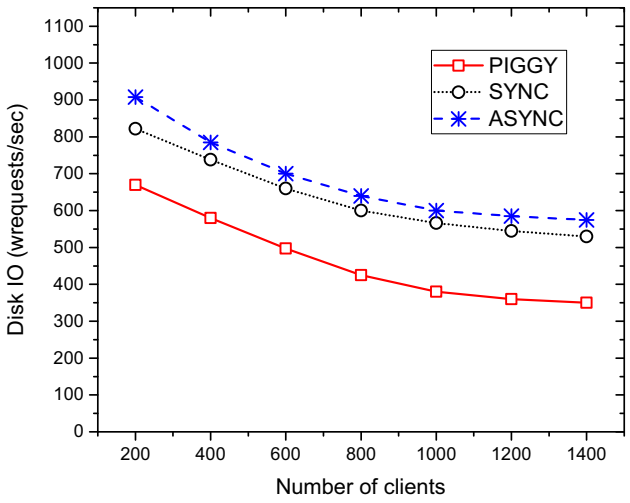

Fig. 5 Write requests over the disk

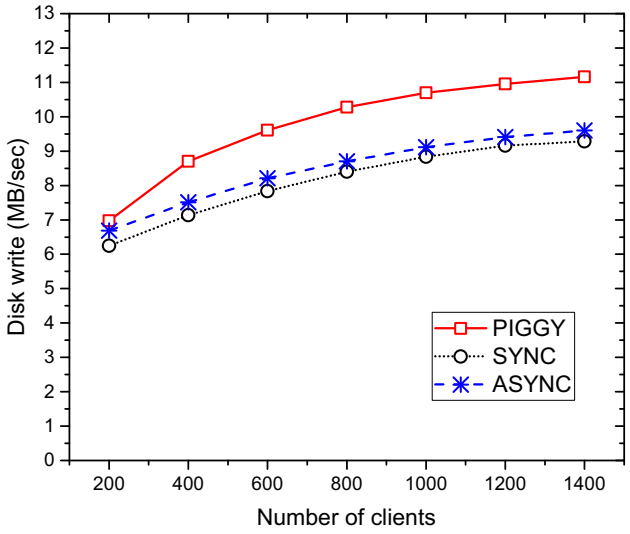

Fig. 6 Write throughput over the disk

shows the transaction throughput of three methods as the workload (number of clients) was scaled up. Note that the PIGGY had the better performance than the other methods in any workload, which significantly demonstrated the effectiveness of our method. Since the PIGGY could not produce additional impact on the disk, it improved throughput of write transactions by at least 1.3× when the number of clients was 1400. And the SYNC and ASYNC had a nearly same *TPS*, because the latency of flushing committed LSN is <2 ms which is largely smaller than a common transaction response delay.

We evaluated *IOPS* by using the Linux tool `iostat` which can monitor the number of write requests issued to the specified device per second. Through the comparison of the three methods, Fig. 5 shows that the Piggybacking method had the lowest write requests per second, because it

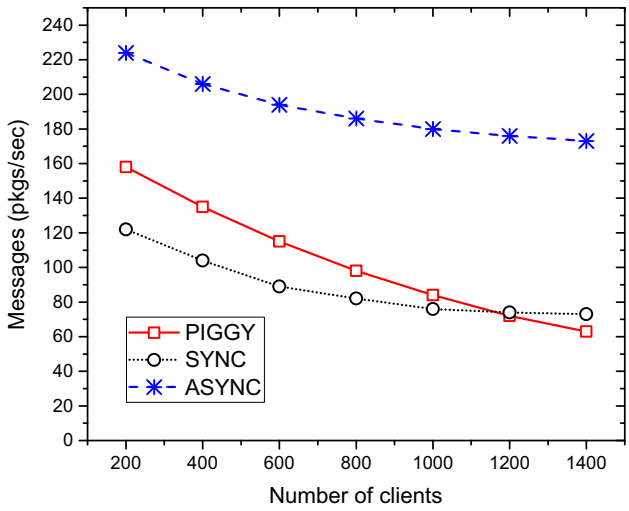

Fig. 7 Message throughput in one follower

needs not to store the commit point, which could incur additional IO and then increase the disk overhead. The ASYNC method had the highest number of write requests, because the single thread flushed the commit point every 10 ms, which caused more requests than SYNC persisted the committed LSN to local disk only after the group of transactional log entries committed. The curves of all the methods had the decreasing trends with the increase in connections, we note that the *commit_interval* described in Sect. 4.3 would be larger as the number of clients increased.

Figure 6 shows the write throughput over the disk of the tree methods. In this case, we also used the `iostat` command to evaluate this performance. Note that PIGGY has the highest write throughput and the ASYNC and SYNC had the similar results. The curves of these mechanisms had the same trend with the ones in Fig. 4, which indicated that the throughput of transactions determined the data volume of disk writes and the flushing committed LSN could not inconspicuously increase the size of data written to the disk. Although the PIGGY adopted a special log to record commit point, it is more efficient than other methods.

We compared the PIGGY with other methods for the case of *Follower receiving* through monitoring the number of packets received by a Follower within one second during normal processing. Figure 7 shows the results of receiving messages per second as the workload (number of clients) was scaled up. The ASYNC method had the highest results since a background thread sent commit point in Leader frequently. With the increase in client connections, all of the three curves decreased by degrees. The reason of the decline is same as the discussion described in Fig. 5. Since the flushing commit point in SYNC increases the processing time of a group of log entries, the SYNC method

has the lowest Follower receiving when the number of client was less than 1000. With the increase in connections, the PIGGY was more effective in group commit mechanism. Therefore, the PIGGY had the lowest Follower receiving when the number of connections was >1200.

From the above experiment results, we could draw a conclusion that the persistence of committed LSN—used to improve the availability of the system—could lead to much more overhead of disk, and then decreases the capacity of IO, which is respected as a precious commodity to replicate and persist transactional log. Therefore, the PIGGY has a better performance than other approaches. Moreover, the SYNC could provide similar throughput of transactions to ASYNC.

7 Conclusion

Log replication based on Paxos can provide database systems with scalability, consistency and highly availability. This paper described an implementation mechanism of Paxos replication for OceanBase, which is scalable and has a memory transactional engine. Unlike traditional implementation, our method takes into account the overhead of storage and network, which have a significant impact on performance.

We find that the synchronization of committed LSN used for timeline consistency may improve the overhead of the system. Therefore, we make use of piggybacking technique to implement log replication and database recovery. Compared to the synchronization mechanism, our method improves throughput of update operations by 1.3×.

Acknowledgements This work is partially supported by National High-tech R&D Program (863 Program) under Grant Number 2015AA015307 and National Science Foundation of China under Grant Number 61332006.

References

1. Baker J, Bond C, Corbett JC et al (2011) Megastore: providing scalable, highly available storage for interactive services. In: Fifth biennial conference on innovative data systems research, pp 223–234
2. Balakrishnan M, Malkhi D, Davis JD et al (2013) CORFU: a distributed shared log. ACM Trans Comput Syst 31(4):10
3. Balakrishnan M, Malkhi D, Wobber T et al (2013) Tango: distributed data structures over a shared log. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles, pp 325–340
4. Bernstein PA, Reid CW, Das S (2011) Hyder—a transactional record manager for shared flash. In: Fifth biennial conference on innovative data systems research, pp 9–20
5. Burrows M (2006) The Chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th symposium on operating systems design and implementation, pp 335–350
6. Cassandra website. http://cassandra.apache.org/
7. Cattell R (2011) Scalable SQL and NoSQL data stores. SIGMOD Rec 39(4):12–27
8. Cooper BF, Ramakrishnan R, Srivastava U et al (2008) PNUTS: Yahoo!'s hosted data serving platform. Proc VLDB Endow 1(2):1277–1288
9. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on cloud computing, pp 143–154
10. Corbett JC, Dean J, Epstein M et al (2013) Spanner: Google's globally distributed database. ACM Trans Comput Syst 31(3):8
11. DeCandia G, Hastorun D, Jampani M, et al (2007) Dynamo: Amazon's highly available key-value store. In: Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles, pp 205–220
12. Dragojević A, Narayanan D, Nightingale EB et al (2015) No compromises: distributed transactions with consistency, availability, and performance. In: Proceedings of the 25th symposium on operating systems principles, pp 54–70
13. Gray J, Helland P, O'Neil P, Shasha D (1996) The dangers of replication and a solution. SIGMOD Rec 25(2):173–182
14. Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system. ACM SIGOPS Oper Syst Rev 44(2):35–40
15. Lamport L (1998) The part-time parliament. ACM Trans Comput Syst 16(2):133–169
16. Lamport L (2001) Paxos made simple. ACM SIGACT News 32(4):18–25
17. Lamport L (2006) Fast paxos. Distrib Comput 19(2):79–103
18. Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P (1992) ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans Database Syst 17(1):94–162
19. OceanBase website. https://github.com/alibaba/oceanbase/
20. Ongaro D, Ousterhout JK (2014) In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX conference on USENIX annual technical conference, pp 305–319
21. Ousterhout J, Agrawal P, Erickson D et al (2010) The case for RAMClouds: scalable high-performance storage entirely in DRAM. SIGOPS Oper Syst Rev 43(4):92–105
22. Patterson S, Elmore AJ, Nawab F, Agrawal D, El Abbadi A (2012) Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. Proc VLDB Endow 5(11):1459–1470
23. Rao J, Shekita EJ, Tata S (2011) Using paxos to build a scalable, consistent, and highly available datastore. Proc VLDB Endow 4(4):243–254
24. Shute J, Vingralek R, Samwel B et al (2013) F1: a distributed SQL database that scales. Proc VLDB Endow 6(11):1068–1079
25. Thomson A, Diamond T, Weng SC et al (2012) Calvin: fast distributed transactions for partitioned database systems. In: Proceedings of the 2012 ACM SIGMOD international conference on management of data, pp 1–12
26. ZooKeeper website. http://zookeeper.apache.org/